Parallelizing Dense and Banded Linear Algebra Libraries using SMPSs

Rosa M. Badia* José R. Herrero† Jesus Labarta* Josep M. Perez*
Enrique S. Quintana-Ortí‡ Gregorio Quintana-Ortí‡

October 29th, 2008
Research Report: UPC-DAC-RR-2008-64

Abstract

The promise of future many-core processors, with hundreds of threads running concurrently, has lead the developers of linear algebra libraries to rethink their design in order to extract more parallelism, further exploit data locality, attain a better load balance, and pay careful attention to the critical path of computation.

In this paper we describe how existing serial libraries like (C-)LAPACK and FLAME can be easily parallelized using the SMPSs tools, consisting of a few OpenMP-like pragmas and a run-time system. In the LAPACK case, this usually requires the development of blocked algorithms for simple BLAS-level operations, which expose concurrency at a finer grain. For better performance, our experimental results indicate that column-major order, as employed by this library, needs to be abandoned in benefit of a block data layout. This will require a deeper rewrite of LAPACK or, alternatively, a dynamic conversion of the storage pattern at run-time. The parallelization of FLAME routines using SMPSs is quite simple as this library includes blocked algorithms (or algorithms-by-blocks in the FLAME argot) for most operations and storage-by-blocks (or block data layout) is already in place.

Keywords: Linear algebra libraries, programmability, high performance, dynamic scheduling, multi-core processors.

1 Introduction

There is a general consensus that with the advent of the new multi-core processors and hardware accelerators as, e.g., the Cell B.E., ClearSpeed boards, or NVIDIA and AMD/ATI GPUs, libraries for dense and banded linear algebra like LAPACK [1], will undergo a complete rewrite to exploit the full potential of these architectures. Optimizing the routines in these libraries is crucial: a vast number of complex scientific and engineering applications can be decomposed into simpler dense and banded linear algebra operations, which usually form the most time-consuming part in these applications. Besides, dense linear algebra has been a traditional battlefield for the priests of high-performance computing and the hardware chip manufacturers, as the LINPACK benchmark and the Top500 list still remind us twice per year.

Indeed, the redesign of high-performance libraries for dense linear algebra is already undergoing as part of the PLASMA and FLAME projects [7, 6, 8, 10, 22, 23, 9, 24] with a similar approach proposed in both cases. In particular, the two projects propose the use of a run-time system to conduct a two-stage execution: An initial symbolic analysis of the code decomposes the computations into sub-operations, or tasks, identifying the dependences among these. This is followed by the actual computation, with a dynamic scheduling mechanism in charge of extracting as much task-parallelism as possible while fulfilling the data dependences

*Barcelona Supercomputing Center – Centro Nacional de Supercomputación (BSC-CNS) and Universitat Politècnica de Catalunya, Nexus II Building, C. Jordi Girona 29, 08034–Barcelona, Spain. {rosa.m.badia, josep.m.perez, jesus.labarta}@bsc.es.

†Depto. de Arquitectura de Computadores, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain. josepr@ac.upc.es.

‡Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain. {quintana, gquintan}@icc.uji.es.

among tasks. The ideas are thus similar to those proposed earlier in the Cilk and the StarSs (GRIDSs, CellSs, SMPSs) projects [5, 2, 20, 18, 19] for problems more general than just dense linear algebra.

In this paper we describe and analyze the parallelization of dense and banded linear algebra libraries using SMPSs (*SMP Superscalar*). The study involves not only the experimental evaluation of the parallel performance but also an analysis of the programmability of the solution, an aspect which is receiving considerable attention in the last years. Specifically, the main contributions of the paper are:

- A demonstration that the parallelization approach proposed in SMPSs is backward compatible with legacy-like code in LAPACK as well as new disruptive approaches like FLAME.
- A presentation focused on both the programmability and the performance of the approach.
- A complete evaluation of the major dense matrix factorizations for the solution of linear systems, including an analysis of the scalability.
- A demonstration that, provided the necessary numerical kernels are available, the same linear algebra codes (with minor modifications) will carry over to platforms like the Cell B.E.

The rest of the paper is structured as follows. In Section 2 we illustrate how SMPSs can be applied to both LAPACK and FLAME to easily produce parallel codes for the solution of linear systems and linear least-squares problems [11]. Experimental results measuring the parallel performance and scalability of the SMPSs codes are reported in Section 3, and a few concluding remarks follow in Section 4.

2 Parallelization of Linear Algebra Algorithms

Linear algebra libraries as LAPACK [1] and FLAME [12, 3, 4] provide numerical routines for the solution of linear systems and linear least-squares problems, and the computation of eigenvalues and singular values of dense (and, for linear systems, banded) matrices. Here we focus on the first two problems and the key numerical methods for their solution: the Cholesky, LU and QR matrix factorizations. Assume the matrices involved in all three problems are square and let n denote their dimension. Common to all three factorizations are their computational cost, $O(n^3)$ floating-point arithmetic operations (flops), and the amount of data involved, $O(n^2)$ numbers. (When the matrix involved in the factorization presents a band structure, the computational and storage costs are reduced, respectively, to $O(k_l k_u n)$ flops and $O((k_l + k_u)n)$ data, where k_l and k_u represent, respectively, the dimensions of the lower and upper bandwidths.) It is the high computational cost of these operations that makes them appealing candidates for parallelization, specially when the matrix dimension is large.

The blocked algorithms described next exploit the cubic cost of the method, and the difference of one order of magnitude with the amount of data involved in the operation, to yield high-performance on current desktop processors with hierarchical memory systems.

2.1 Cholesky factorization of dense matrices

Consider first the Cholesky factorization of a dense symmetric positive definite (SPD) matrix $A \in \mathbb{R}^{n\times n}$, defined as

$$A = LL^T, \tag{1}$$

where $L \in \mathbb{R}^{n\times n}$ is a lower triangular matrix known as the Cholesky factor of A. (Alternatively, A can be decomposed as $A = U^T U$, with $U \in \mathbb{R}^{n\times n}$ being upper triangular.)

Blocked algorithms for the Cholesky factorization consist of a single loop that traverses the matrix from the top left corner to the bottom right one, processing several rows/columns of the matrix per iteration. Inside the loop, operations are performed to compute a certain part of the triangular factor and/or update other parts (blocks) of the matrix. In practice, the algorithm overwrites the lower triangular part of A with the entries of L, while the strictly upper triangular part remains unmodified.

Figure 1 presents a fragment of a (C)LAPACK-like routine `dpotrf`, which implements a right-looking algorithm for the Cholesky factorization of an `n` $\times$ `n` SPD matrix (with double-precision real entries), stored in column-major order in an array `A` with leading dimension `lda`. At each iteration of the loop, a `jb` $\times$ `jb`

```
int dpotrf(char *uplo, int n, double *A, int lda, int nb, int *info)
{
/* Blocked algorithm (right-looking variant) to compute the lower
   triangular Cholesky factorization A = L*L^T, with algorithmic block size nb. */

   static double done = 1.0;
   static double minus_done = -1.0;
/* ... */

   for (j = 1; j <= n; j += nb) {

/*   Factorize the current diagonal block.  */
     jb = min(nb, n - j + 1);
     dpotf2(uplo, &jb, &A_ref(j, j), &lda, info);
/*   ... */

     if(j + jb <= n) {

/*     Compute the current block subdiagonal block.  */
       i__3 = n - j - jb + 1;
       dtrsm("Right",     "Lower",
             "Transpose", "Non-unit",
             &i__3,       &jb,
             &done,       &A_ref(j, j),      &lda,
                          &A_ref(j + jb, j), &lda);

/*     Update the bottom right block.  */
       dsyrk("Lower",     "No transpose",
             &i__3,       &jb,
             &minus_done, &A_ref(j + jb, j),      &lda,
             &done,       &A_ref(j + jb, j + jb), &lda);
     }
   }
/* ... */
}
```

Figure 1: (C)LAPACK-like blocked algorithm for computing the Cholesky factorization.

block on the diagonal of the matrix is factorized using an unblocked routine `dpotf2`, the corresponding $\texttt{k} \times \texttt{jb}$ subdiagonal block of the Cholesky factor, with $\texttt{k} = \texttt{i\_\_3} = \texttt{n} - \texttt{j} - \texttt{jb} + 1$, is computed using the triangular solver `dtrsm` in BLAS, and the $\texttt{k} \times \texttt{k}$ block to its right is updated using the BLAS kernel `dsyrk` for the symmetric rank-`k` update.

Figure 2 shows an equivalent algorithm expressed using the object-oriented notation in FLAME (left) and the corresponding routine encoded using the FLAME/C Application Programming Interface (right) [3, 4]. In the algorithm, $m(B)$ stands for the number of rows of B and $A := \{L\backslash A\}$ denotes that the lower triangular part of A is overwritten by L while the strictly upper triangular part remains unmodified. We believe the rest of notation to be intuitive. Routines `FLA_Chol_unb_var1`, `FLA_Trsm`, and `FLA_Syrk` in the FLAME code play analogous roles to `dpotf2`, `dtrsm`, and `dsyrk` in Figure 1, respectively.

2.2 Traditional parallelization

The conventional approach to extract parallelism from LAPACK and FLAME codes on shared-memory multiprocessors (including SMPs, NUMA platforms, and multi-core processors) has been to pass all the burden for the parallel execution to the BLAS so that the LAPACK/FLAME codes remain unchanged. While the existence of efficient multi-threaded implementations of BLAS provides the means for a straight-forward, transparent parallelization of these libraries, the performance delivered by this approach is suboptimal. The fork-join model of execution of this strategy, where all threads synchronize at the beginning and the completion of each BLAS call, introduces a non-negligible overhead and imposes artificial constraints on the parallelism of the operation. Look-ahead techniques have been proposed in the past to increase the parallelism and scalability of this solution, but they do so at the cost of a much more complicated coding [25].

2.3 Dynamic parallelization with SMPSs

As an alternative to the parallelization strategy based on the use of multi-threaded BLAS and the look-ahead techniques, Cilk and SMPSs in general, and LAPACK/PLASMA and FLAME/SuperMatrix for dense

```
Algorithm: $A := \text{CHOL\_BLK\_VAR1}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do
    Determine block size $b$
    Repartition
    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
        where $A_{11}$ is $b \times b$

    $A_{11} := \{L \backslash A\}_{11} = \text{CHOL\_UNB}(A_{11})$
    $A_{21} := L_{21} = A_{21} L_{11}^{-\mathrm{T}}$
    $A_{22} := A_{22} - L_{21} L_{12}^{\mathrm{T}} = A_{22} - A_{21} A_{21}^{\mathrm{T}}$

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

```c
FLA_Error FLA_Chol_blk_var1( 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_var1( 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;
}
```

Figure 2: FLAME blocked algorithm for computing the Cholesky factorization (left) and the corresponding FLAME/C implementation (right).

linear algebra, propose to (semi-)automatically decompose the problem into tasks, identify dependencies among these, and dynamically issue ready tasks to be executed on the processors (or cores) of the system. When applied to dense linear algebra, this strategy has demonstrated a clear advantage over the traditional approach in terms of performance. Combined with the high-level approach of FLAME, it also offers an elegant solution for the programmability problem, overcoming the difficulties posed by look-ahead techniques.

In this subsection we describe how SMPSs can be used to parallelize traditional codes in LAPACK as well as object-oriented FLAME codes. The result is a library of parallel codes that employs the SMPSs run-time system, replacing similar mechanisms being developed as part of the PLASMA (for LAPACK) and SuperMatrix (for FLAME) extensions.

The basic components of SMPSs are an OpenMP-like directive to identify units of execution, or tasks, and the run-time system tailored for multi-core environments. The *task construct* directive:

```
# pragma css task [clause [clause] ...]
  function-definition | function-declaration
```

inserted before a C function, declares it as a task, candidate to be executed in one of the processors (or cores) of the system concurrently with tasks running in other processors. The programmer implicitly exposes data dependencies in the operation to the run-time system by passing additional information via the *task construct clauses*:

```
input  (parameter-list)
output (parameter-list)
inout  (parameter-list)
```

which serve to indicate which parameters to the function are only read, only written, or read and written by the task. Then, during the execution, the run-time system analyzes data dependencies among the tasks to determine which can be issued to execution.

Armed with this directive, we can proceed to parallelize the LAPACK and FLAME codes for the Cholesky factorization as described in the next subsections.

2.3.1 LAPACK

Let us start with the (C)LAPACK-like code in Figure 1. We could naively assume that including task construct directives with the appropriate clauses before the header definition for `dpotf2`, `dtrsm`, and `dsyrk` suffices. However, given the data dependencies between the operations in one iteration of the loop (`dsyrk` depends on the result of `dtrsm`, which in turn depends on the factorization performed in `dpotf2`), this results in a full serial execution. Therefore, we need to identify a finer-grain parallelism. We can do so by decomposing the triangular system solve and the symmetric rank-`k` update into sub-operations (finer-grain tasks) which can proceed in parallel. Now, these two operations belong to BLAS, not to LAPACK, so that we need to design blocked algorithms for them. Figure 3 shows blocked algorithms `dtrsm_by_blocks` and `dsyrk_by_blocks` for the particular shapes of the triangular system solve and the symmetric rank-`k` update, respectively, appearing in the Cholesky factorization. Arguments `mb` and `nb` correspond to the algorithmic block size, and determine the granularity of the decomposition of the operation into tasks of finer grain. Routines `dtrsm_tile`, `dsyrk_tile`, and `dgemm_tile` are simple wrappers to the corresponding routines from BLAS. For instance, the code for `dtrsm_tile` is as follows:

```
void dtrsm_tile(int    m,     int    n,
                double alpha, double *A,   int lda,
                              double *B,   int ldb)
{
     dtrsm("Right",     "Lower",
           "Transpose", "Non-unit",
           &m,          &n,
           &alpha, A,   &lda,
                   B,   &ldb);
}
```

Given these two blocked algorithms, we need to substitute the calls to `dtrsm` and `dsyrk` in Figure 1 by `dtrsm_by_blocks` and `dsyrk_by_blocks`, respectively, as well as replace the call to `dpotf2` by a call to a wrapper routine `dpotrf_tile` which simply invokes routine `dpotrf`. (In our experiments we found out that, for the optimal task size, it is convenient to invoke the BLAS-3 routine to factorize the diagonal blocks instead of the BLAS-2 routine.) We will finally need to add the following *SMPSs directives to the header declaration of the wrapper routines*:

```
#pragma css task input (uplo, n, lda)\
                 inout (A[1], info[1])
void dpotrf_tile(int n, double *A, int lda, int *info);

#pragma css task input (m, n, alpha, A[1], lda, ldb)\
                 inout (B[1])
void dtrsm_tile(int    m,     int    n,
                double alpha, double *A, int lda,
                              double *B, int ldb);

#pragma css task input (n, k, alpha, A[1], lda, beta, ldc)\
                 inout (C[1])
void dsyrk_tile(int    n,     int    k,
                double alpha, double *A, int lda,
                double beta,  double *C, int ldc);

#pragma css task input (m, n, k, alpha, A[1], lda, B[1], ldb, beta, ldc)\
                 inout (C[1])
void dgemm_tile(int    m,      int    n, int k,
                double alpha,  double *A, int lda,
                               double *B, int ldb,
                double beta,   double *C, int ldc);
```

With these declarations, at execution the run-time system identifies each call to one of the previous routines as a single task, with the granularity of the tasks being determined by the algorithmic block size. The directionality of the operands (input, input/output, or output) and the actual values of these parameters determine the dependencies among the tasks. For the Cholesky factorization, as well as for many other

```
void dtrsm_by_blocks(int    m,      int     n,
                     double alpha, double *A, int lda,
                                   double *B, int ldb,
                     int    mb)
{
/* Blocked algorithm to solve a lower triangular linear
   system X = B*A^-T, with algorithmic block size mb.
   Assumptions:
        A consists of one block and
        B consists of a column of blocks. */

   static double done = 1.0;
/* ... */

   for (i = 1; i <= m; i += mb) {

/*   Solve triangular system for current block. */
     ib = min(mb, m - i + 1);
     dtrsm_tile(ib,    n,
                alpha, A,           lda,
                       &B_ref(i, 1), ldb);
   }
/* ... */
}
```

```
void dsyrk_by_blocks(int    n,     int    k,
                     double alpha, double *A, int lda,
                     double beta,  double *C, int ldc,
                     int    nb)
{
/* Blocked algorithm to compute a (lower) symmetric rank-k
 * update C := C + A*A^T, with algorithmic block size nb.
   Assumption:
        A consists of a column of blocks. */
/* ... */

   for (i = 1; i < n; i += nb) {
/*   Update current diagonal block. */
     ib = min(nb, n - i + 1 );
     dsyrk_tile(ib,     k,
                alpha, &A_ref(i, 1),   lda,
                beta,  &C_ref( i, i ), ldc );

     for (j = 1; j <= i-1; j += nb ) {
/*     Update blocks to the right. */
       jb = min(nb, i-j);
       dgemm_tile(ib,     jb,            k,
                  alpha, &A_ref( i, 1 ), lda,
                         &A_ref( j, 1 ), lda,
                  beta,  &C_ref( i, j ), ldc );
     }
   }
/* ... */
}
```

Figure 3: CLAPACK-like blocked algorithms for the triangular system solve (left) and the symmetric rank-k update (right).

dense and banded linear algebra operations, blocks do not overlap. This is extremely useful as it allows us to determine data dependencies based solely on the initial address of each block. Thus, e.g., the run-time system can detect a *Read-After-Write* (RAW) dependency between the Cholesky factorization of a diagonal block (call to `dpotrf_tile`) and the triangular system solves for each one of the corresponding subdiagonal blocks by just comparing the address of the (1,1) element in the diagonal block (`&A_ref(j,j)` for iteration `j`) with that of the first matrix argument of the call to `dtrsm_tile`.

Also, the only dependencies that appear during the Cholesky factorization are of type RAW, and the runtime does not employ renaming to eliminate false dependencies.

While, in principle, one could annotate directly the interfaces of the LAPACK routine `dpotf2` and BLAS `dtrsm`, `dsyrk`, and `dgemm`, using the wrappers is a simple safeguard against unexpected tasks being created due to other calls to these routines encountered in the code.

2.3.2 FLAME

The strategy to follow with the FLAME code is similar. We first need to decompose `FLA_Trsm` and `FLA_Syrk` into smaller sub-operations by invoking instead blocked algorithms for these, `FLA_Trsm_by_blocks` and `FLA_Syrk_by_blocks`, respectively. Figure 4 provides an algorithm-by-blocks and the corresponding FLAME/C code for the former while the latter presents a similar structure.

The fragment of code in Figure 5 illustrates what happens inside `FLA_Trsm_tile`: object properties as the size and leading dimension are first extracted via functions/macros which is followed by the invocation of BLAS routine `dtrsm`; we note here that `FLA_Obj_buffer` is just a function/macro that extracts the address of the first element of data block encapsulated in the object. The figure thus shows that, in order to parallelize the FLAME code, we only need to replace the calls to `dpotf2`, `dtrsm`, `dsyrk` and `dgemm` by calls to the appropriate wrapper routines (`dpotrf_tile`, `dtrsm_tile`,…), and insert the *same SMPSs directives to the header declarations of the wrappers* as were specified for LAPACK in the previous subsection.

Note that encapsulation/abstraction at the level employed in FLAME plays a major role in keeping code and data storage independent as, e.g., nothing prevents the data involved in the problem from being stored in column-major order or following more advanced (and also convenient) patterns like storage-by-blocks. This is all transparent to the code and it is only determined by the manner in which data is initially inserted

```
Algorithm: $[B] := \text{FLA\_TRSM\_BY\_BLOCKS}(A, B)$
Partition $B \rightarrow \left(\frac{B_T}{B_B}\right)$
  where $B_T$ has 0 rows
while $m(B_T) < m(B)$ do
  Determine block size $b$
  Repartition
    $\left(\frac{B_T}{B_B}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right)$
      where $B_1$ has $b$ rows
  ------------------------------
  $B_1 := B_1 A^{-T}$
  ------------------------------
  Continue with
    $\left(\frac{B_T}{B_B}\right) \leftarrow \left(\begin{array}{c} B_0 \\ B_1 \\ \hline B_2 \end{array}\right)$
endwhile
```

```
void FLA_Trsm_by_blocks( FLA_Obj alpha, FLA_Obj L,
                                        FLA_Obj B,
                         int     mb_alg )
/* Blocked algorithm to solve a lower triangular linear
   system X := B*A^-T, with algorithmic block size mb.
   Assumptions:
        A consists of one block and
        B consists of a column of blocks. */
{
  FLA_Obj BT,      B0,
          BB,      B1,
                   B2;
  int b;

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

  while ( FLA_Obj_length( BT ) < FLA_Obj_length( B ) ) {
    b = min( FLA_Obj_length( BB ), mb_alg );
    FLA_Repart_2x1_to_3x1( BT,       &B0,
                        /* ** */   /* ** */
                                     &B1,
                           BB,       &B2,    1, FLA_BOTTOM );
    /*---------------------------------------------*/
    FLA_Trsm_tile( alpha, A,
                          B1 );
    /*---------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &BT,        B0,
                                          B1,
                            /* ** */   /* ** */
                              &BB,        B2,     FLA_TOP );
  }
}
```

Figure 4: FLAME algorithm-by-blocks for the solution of a triangular system (left) and the corresponding FLAME/C implementation (right).

```
void FLA_Trsm_tile( FLA_Obj alpha, FLA_Obj A,
                                   FLA_Obj B )
{
/* ... */
  m_A      = FLA_Obj_length  ( A );
  n_A      = FLA_Obj_width   ( A );
  ldim_A   = FLA_Obj_ldim    ( A );

  m_B      = FLA_Obj_length  ( B );
  n_B      = FLA_Obj_width   ( B );
  ldim_B   = FLA_Obj_ldim    ( B );

/* ... */
  dtrsm( "Right",                                  "Lower",
         "Tranpose",                               "Non-unit",
         &m_B,                                     &n_B,
         ( ( double * ) FLA_Obj_buffer( alpha ) ), ( ( double * ) FLA_Obj_buffer( A ) ), &ldim_A,
                                                   ( ( double * ) FLA_Obj_buffer( B ) ), &ldim_B );
/* ... */
}
```

Figure 5: Inside the FLAME/C implementation of routine `FLA_Trsm_tile` for the solution of a triangular linear system.

in the data structure.

Results in section 3 will report the practical parallel performance attained by SMPSs.

2.4 Other dense matrix factorizations: LU and QR

The elaboration in the previous subsection has shown how easy is to parallelize the code for the Cholesky factorization in the (C)LAPACK or FLAME libraries using SMPSs. The same approach carries over to more complex matrix operations for the solution of linear systems as the LU and QR factorizations. By simply creating wrappers to the underlying LAPACK/BLAS calls, and annotating these with the appropriate SMPSs directives, we would obtain a parallel execution orchestrated by the SMPSs run-time system. A key difference between these two decompositions and the Cholesky factorization, the presence of WAR/WAW (write-after-read/write-after-write), is addressed by the run-time system employing a renaming technique transparent to the programmer.

A major obstacle to the efficient parallelization of the traditional LU and QR factorization remains: the need to factorize a complete block of columns before the corresponding transformations can be applied to the remaining part of the matrix, serializes the execution of the algorithm and drastically reduces the amount of parallelism. For the LU factorization, this requisite is imposed by the use of partial pivoting as, for each column, the diagonal element may be swapped with any of the subdiagonal entries of the entire column. For the QR factorization, this is due to the use of Householder transformations which involve the diagonal and all subdiagonal elements in each column.

The introduction of an out-of-core (OOC) algorithm in [13] for the computation and updating of the QR factorization of large-scale problems provided a more scalable solution for this operation. The basic idea is to compute a "tiled" QR factorization, where the matrix is decomposed into small square blocks, or *tiles*, and each subdiagonal block is factorized with respect to the diagonal block independently from others. This variant exhibits more parallelism as, e.g., once the $(k+1,k)$ block is factorized, the updates of those blocks to the right can be updated in parallel with the factorization of blocks $(k+2,k)$, $(k+3,k)$, etc. and the corresponding updates of the blocks to their right. The application of this technique to multi-core systems was first proposed in [7], but incurred in a non-negligible computational overhead. In [23] a second variant which mimicked the two levels of blocking in [13], reduced the cost of the tiled QR factorization on multi-core processors to that of the traditional one, making it a practical approach. The traditional and tiled algorithms only employ orthogonal transformations and, therefore, the numerical stability of all implementations is equivalent.

The pursuit of a tiled LU factorization with pivoting for OOC computations has been a long quest as the need to preserve pivoting stands in the way of performance. This was solved in [15, 21] by proposing a modified pivoting scheme, named as incremental pivoting, which enabled high performance while maintaining a numerical stability close to that of partial pivoting. The idea here is analogous to that of the QR factorization: factorize each subdiagonal block independently so that the updates of the blocks to the right can proceed concurrently with the factorization of other subdiagonal blocks. Here partial pivoting is applied between pairs of blocks leading to a blocked version of pairwise pivoting. Again, the algorithm was initially designed for OOC and as a solution to the problem of updating an LU factorization, but its extension to the multi-core arena is straight-forward. Both PLASMA and FLAME have recently evaluated the parallelism of the tiled LU factorization for multi-core systems [6, 22].

Let us elaborate to some extent the parallelization of the tiled QR factorization using SMPSs. Assume we are given a serial C code for the computation of the tiled QR factorization of an `n` $\times$ `n` matrix stored in an array `A` like that in Figure 6. (This code is not part of the standard distributions of LAPACK or `libflame`, but experimental implementations have been developed as part of the extensions of the two projects for multi-core processors. For simplicity, many of the arguments have been hidden in the code.) In order to parallelize this code with SMPSs, we would only need to create wrappers for `dgeqrf`, `dormqr`, `dgeqrf_2x1`, `dormqr_2x1`, and add the corresponding pragmas to identify the calls to these wrappers as the unit of computation (tasks) specifying the directionality of the arguments. For example, for routine `dgeqrf`, the header for the corresponding wrapper could be declared as follows:

```
#pragma css task input (m, n, lda)\
                 inout (A[1], tau[1], info[1])
void dgeqrf_tile(int m, int n, double *A, &lda, double *tau, int *info);
```

while the other routines would present similar declarations. Again, this example shows how straight-forward is to translate an existing sequential implementation into a parallel one using the SMPSs infrastructure. The algorithm for the tiled LU factorization presents a much similar structure and can be annotated with SMPSs directives in an analogous manner.

The experimental results in section 3 will demonstrate the superior parallelism and scalability inherent to the tiled algorithms for the LU and QR factorization, and the efficacy of SMPSs in identifying and exploiting these properties.

2.5 Storage-by-blocks

Storing the elements of matrices by blocks (also known as block data layout, storage-by-blocks, or hierarchical storage) has been experimentally demonstrated to increase data locality and reduce the number of TLB misses; see, e.g., [17]. The basic idea is to lay the elements in a block in consecutive positions of the memory

```
void dgeqrf_by_blocks(int n, double *A, double *tau, int lda, int nb, int *info)
{
/* Tiled algorithm (right-looking variant) to compute the
   QR factorization A = Q*R, with algorithmic block size nb. */

/* ... */

   for (k = 1; k <= n; k += nb) {

/*   Factorize the current diagonal block.  */
     kb = min(nb, n - k + 1);
     dgeqrf(&kb, &kb, &A_ref(k, k), &lda, &tau(k), info);
/*   ... */

/*   Update blocks to the right of the diagonal block. */
     for (j = k+kb; k <= n; k += nb) {
       jb = min(nb, n - j + 1);
       dormqr("Left", "Transpose",
              &kb, &jb, &kb, &A_ref(k, k), &lda, /* ... */
                             &A_ref(k, j), &lda, /* ... */ );
     }

     for( i = k+kb; i <= n; i += nb ) {
/*     Compute the current subdiagonal block.  */
       ib = kb+min(nb, n - i + 1);
       dgeqrf_2x1(&ib, &kb, &A_ref(k, k), &lda,
                           &A_ref(i, k), &lda, /* ... */ );

       for (j = k+kb; k <= n; k += nb) {
         jb = min(nb, n - j + 1);
/*       Update the blocks to the right of the diagonal and
         current subdiagonal blocks.  */
         dormqr_2x1("Left", "Transpose",
                    &ib, &jb, &kb, &A_ref(i, k), &lda, /* ... */
                                   &A_ref(k, j), &lda, /* ... */
                                   &A_ref(i, j), &lda, /* ... */ );
       }
     }
   }
/* ... */
}
```

Figure 6: (C)LAPACK-like tiled algorithm for computing the QR factorization.

in column- or row-major order to improve data locality. Storage-by-blocks also favors the use of micro-kernels to perform BLAS operations on small blocks, which can increase the performance by reducing the number of times blocks are packed and unpacked [14].

Reconsider now the code for the tiled QR factorization in Figure 6. While we assumed there that the matrix was stored columnwise, as it is usual in LAPACK and Fortran codes, nothing really prevented the matrix from being stored block-wise. To provide correct access to the data though, one would need to define C macros which translate references of the form `A_ref(i, j)` into the appropriate address of the memory and replace the leading dimension of the matrix, `lda`, by the leading dimension of the block.

Although simple, this solution shifts the burden of filling the data for the matrix to the user. One approach around this problem would be to perform the transformation from columnwise storage to block-wise inside the routine, before the computation starts, and undo the transformation at the end. While possible, this solution presents a certain overhead (non-negligible for small matrices), and can be specially inefficient if the matrix is employed in several consecutive operations. Furthermore, for non-square matrices, performing this transformation in-place is not trivial. A second possibility is implicit in the high-level approach provided by FLAME application programming interfaces (APIs). Here, the matrix is viewed as a matrix of matrices so that the user can easily manipulate matrices stored by blocks [16].

2.6 Banded matrix factorizations

Consider now the factorization of a matrix which presents a wide bandwidth. (For narrow band matrices, the computational burden is so small that it is probably worthless to use a parallel computer.) A simple idea will allow us to employ the dense codes to compute the factorization of this matrix incurring only in negligible computational and storage costs overheads compared with the traditional band codes. Let us come back to the code for the dense QR factorization in Figure 6, where we will consider now that `A` is

Platform	Architecture	Frequency (GHz)	L2 cache (KBytes)	L3 cache (MBytes)	RAM per node	CPUs per node (GBytes)
SGI-1	Intel Itanium2	1.5	256	4096	4	2
SGI-2	Intel Itanium2	1.6	256	4096	16	2 dual core

Platform	Linux O.S. kernel	Compiler	Optimization flags	BLAS
SGI-1	2.6.5-7.244-sn2	gcc 3.3.3	`-O3`	MKL 8.1
SGI-2	2.6.16.46-0.12-default	gcc 4.1.2	`-O3`	MKL 9.1

Table 1: Architecture (top) and software (bottom) employed in the evaluation.

laid out block-wise, with only those blocks of the matrix which contain one or more nonzero elements being actually stored. Thus, a reference `A_ref(i, j)` will point into the appropriate direction of the memory if this block contains a nonzero value while it will be a pointer to `NULL` otherwise. Finally, we only need to add a simple sentinel code in the wrapper routines which checks all blocks involved in the operation, immediately returning control in case any of these pointers is void. For example, the wrapper for `dgeqrf_2x1` would be as follows:

```
void dgeqrf_2x1_tile(int    m,  int n,
                     double *A, int lda,
                     double *B, int ldb, /* ... */)
{
    if (A == NULL) || (B == NULL) return 0;

    dgeqrf_2x1(&m, &n,
               A,  &lda,
               B,  &ldb, /* ... */);
}
```

2.7 Other architectures: Cell B.E.

SMPSs is just but a case of the StarSs programming model. This approach has been successfully applied from very large systems like the Grid, in GRIDSs, to much smaller ones like the Cell B.E., as part of CellSs. In the latter case, given efficient (vectorized) kernels to perform the basic LAPACK and BLAS operations defined as tasks on a single SPU, the same annotated codes (with minor modifications) combined with the CellSs run-time provide a parallel solution for the Cell B.E.

3 Experiments

All experiments in this section were performed using double-precision floating-point arithmetic on two SGI Altix multiprocessors. Details on the platforms that were employed in the experimental evaluation are given in Table 1. These systems present a CC-NUMA architecture, with a local RAM being shared by the CPUs in each node, and a SGI NUMAlink interconnection ring. SGI-1 has 16 CPUs running at 1.5 GHz, for a peak performance of 96 GFLOPS (96×10^9 flops per second), while SGI-2 consists of 16 dual core CPUs at 1.6 GHz and the peak performance is 204.8 GFLOPS. Performance was measured in the experiments by linking to the BLAS in Intel Math Kernel Library (MKL). A varied range of block sizes were evaluated in each experiment, but only the results corresponding to the best block size are reported.

The first experiment reports the performance of the (C)LAPACK-like and FLAME routines for the (computation of the lower triangular factor of the) Cholesky factorization, parallelized using SMPSs as described in subsections 2.3.1 and 2.3.2, respectively. Figure 7 reports that the FLAME routine yields superior performance and scalability on SGI-1. This is mainly due to the use of a block data layout in the FLAME routines vs the traditional columnwise employed by the (C)LAPACK-like code. Further experimentation showed that both codes attain basically the same performance if the two codes are modified to use the same data layout.

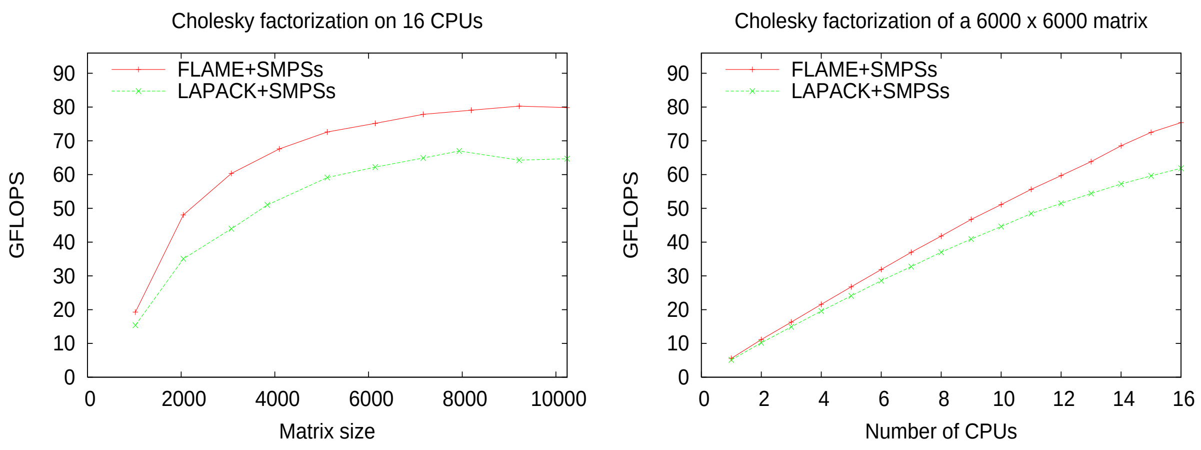

Figure 7: Parallel performance (left) and scalability (right) of the (C)LAPACK-like and FLAME Cholesky factorization routines parallelized with SMPSs.

Our second experiment illustrates the performance of tiled routines combined with storage-by-blocks written in a (C)LAPACK-like style for all three major factorizations. The codes are parallelized and executed using SMPSs on SGI-2. To asses the performance, we include in the results the performance of Intel MKL 9.1 routine for the corresponding factorization: Cholesky (computation of the lower triangular factor), LU with partial pivoting, and traditional QR. Figure 8 reports the parallelism and scalability of these codes. In all three cases, the parallel codes obtained with SMPSs deliver performances much higher than those obtained with the MKL routines. For the LU factorization, we also include the performance of the algorithm with partial pivoting parallelized with SMPSs. The results for the LU and QR factorizations confirm that much of the benefit of the SMPSs parallelizations of these codes comes from the higher degree of parallelism intrinsic to the tiled algorithms. The plots on the right also show a decrease of the scalability of the algorithms as more processors are added. This is partially due to the matrix being created by a single node so that all data remain local to a single node. A "distributed" creation of the matrix would have rendered a higher degree of scalability for all solutions.

4 Conclusions

We are living an explosion in the number of "solutions" (in the form of parallelizing compilers, parallel libraries and languages) to program current multi-core processors and hardware accelerators, and get ready for the future many-core processors. The situation resembles much that of the 90s, with the spread of clusters of workstations among the scientific community, and the arise of a multitude of parallel solutions to program distributed-memory (message-passing) architectures. From what we have experienced in the past, programmability (in terms of flexibility and ease of use) will be the key that determines which among these solutions will last.

In this paper, we have described the use of the SMPSs infrastructure to parallelize existing standard numerical linear algebra libraries like LAPACK and FLAME. The SMPSs tools free the library developer (usually an engineer or a numerical analyst) from the burden of parallelizing a complex code, so that he can focus on developing new algorithms with better numerical properties, lower computational cost, or higher implicit degree of parallelism. In particular, the programmer only needs to provide simple annotations to the sequential code, which give the SMPSs run-time system hints on the granularity of the tasks and the basic information to detect data dependencies at execution.

For dense linear algebra, we have demonstrated that SMPSs efficiently exploits task parallelism, yielding high productivity (in terms of fast parallelization of the codes) and performance for the major factorizations involved in the solution of linear systems. The results carry over to all level 3 BLAS, and likely to those two-

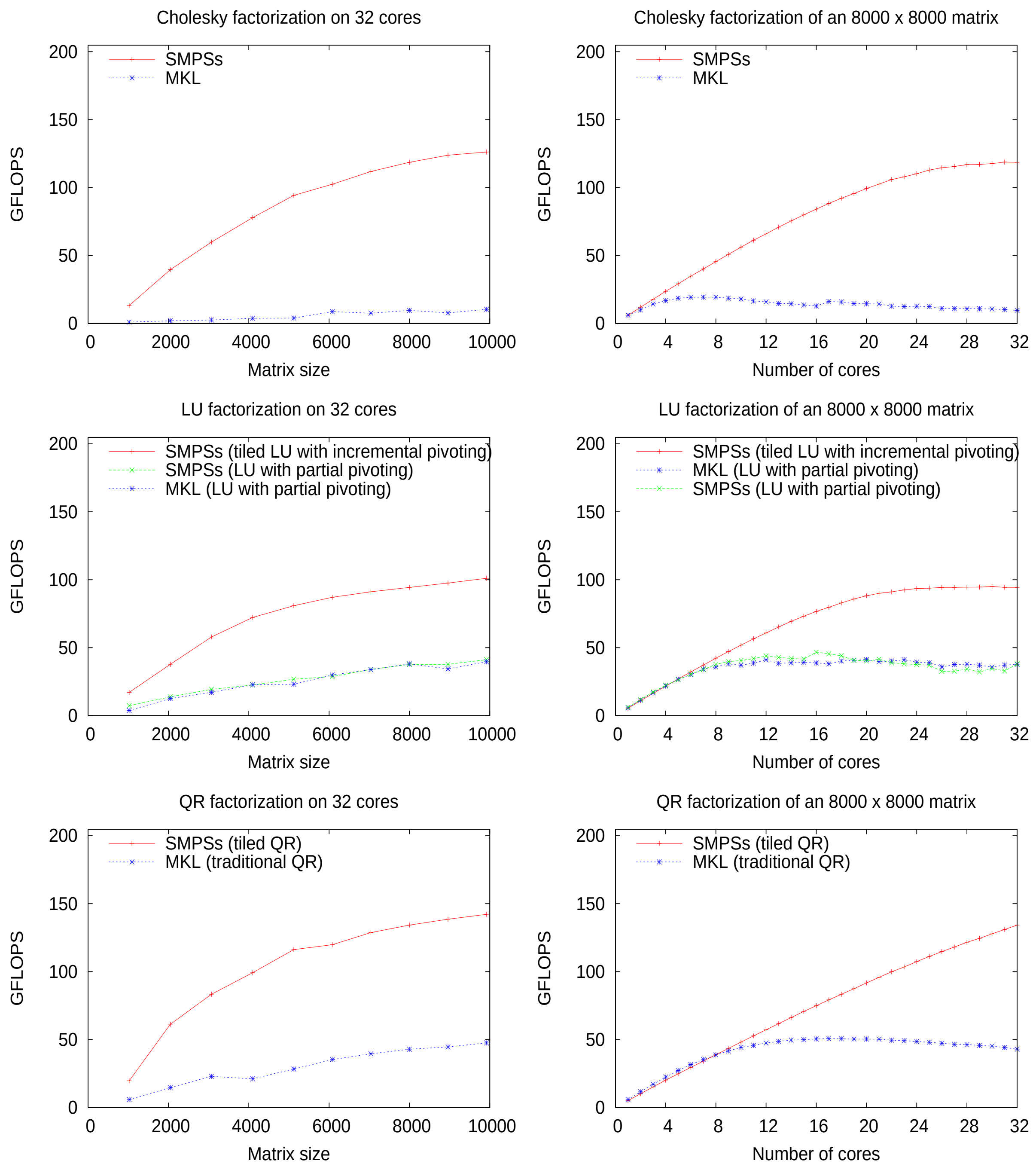

Figure 8: Parallel performance (left) and scalability (right) of the Cholesky (top) LU (middle) and QR (bottom) factorization routines parallelized with SMPSs.

sided reductions to condensed forms for the eigenvalue and singular values problems which can be formulated in terms of level 3 BLAS.

Acknowledgments

The researchers at the BSC-CNS and the Universitat Politècnica de Catalunya acknowledge the financial support of the *Comisión Interministerial de Ciencia y Tecnología* (CICYT, Contract TIN2007-60625) and the BSC-IBM MareIncognito research agreement.

The researchers at the Universidad Jaime I were supported by projects CICYT TIN2005-09037-C02-02 and FEDER, and P1B-2007-19 of the *Fundación Caixa-Castellón/Bancaixa* and UJI. Additional support came from the *Spanish ICTS program* of the Barcelona Supercomputing Center.

We thank Lluís Martinell from the BSC for his help during the parallelization of the LAPACK codes.

References

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

[2] Pieter Bellens, Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. CellSs: a programming model for the Cell BE architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM Press.

[3] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*, 31(1):1–26, March 2005.

[4] 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.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996.

[6] 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.

[7] 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.

[8] 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 *Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2007)*, pages 116–125, San Diego, CA, USA, June 9-11 2007a.

[9] 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)*, pages 123–132, 2008.

[10] 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.

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

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

[13] Brian C. Gunter and Robert A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Trans. Math. Soft.*, 31(1):60–78, March 2005.

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

[15] 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.

[16] Tze Meng Low and Robert van de Geijn. An api for manipulating matrices stored by blocks. Technical Report TR-2004-15, Department of Computer Sciences, The University of Texas at Austin, May 2004.

[17] 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.

[18] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A flexible and portable programming model for SMP and multi-cores. Technical Report 03/2007, Barcelona Supercomputing Center - Centro Nacional de Supercomputación, Barcelona, Spain, 2007.

[19] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A dependency-aware task-based programming environment for multi-core architectures. In Causal Productions, editor, *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 142–151, September 2008. IEEE Catalog Number CFP08235-CDR.

[20] Josep M. Pérez, Pieter Bellens, Rosa M. Badia, and Jesús Labarta. CellSs: Programming the Cell/B.E. made easier. *IBM J. of Research & Development*, 51(5), 2007.

[21] Enrique S. Quintana-Ortí and Robert van de Geijn. Updating an LU factorization with pivoting. *ACM Trans. Math. Soft.*, 2008. To appear.

[22] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Robert van de Geijn, and Field G. Van Zee. Design of scalable dense linear algebra libraries for multithreaded architectures: the LU factorization. In *Workshop on Multithreaded Architectures and Applications – MTAAP 2008*, 2008. CD-ROM.

[23] 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 F. Spies D. El Baz, J. Bourgeois, editor, *16th Euromicro International Conference on Parallel, Distributed and Network-based Processing – PDP 2008*, pages 301–310, 2008.

[24] 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.

[25] 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.