CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **24**:305–321
Published online 22 August 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1829

Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures

Azzam Haidar [1], Hatem Ltaief [1,*,†], Asim YarKhan [1] and Jack Dongarra [1,2,3]

[1]*Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, USA*
[2] *Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennesse*
[3]*School of Mathematics & School of Computer Science, University of Manchester*

SUMMARY

The objective of this paper is to analyze the dynamic scheduling of dense linear algebra algorithms on shared-memory, multicore architectures. Current numerical libraries (e.g., linear algebra package) show clear limitations on such emerging systems mainly because of their coarse granularity tasks. Thus, many numerical algorithms need to be redesigned to better fit the architectural design of the multicore platform. The parallel linear algebra for scalable multicore architectures library developed at the University of Tennessee tackles this challenge by using tile algorithms to achieve a finer task granularity. These tile algorithms can then be represented by directed acyclic graphs, where nodes are the tasks and edges are the dependencies between the tasks. The paramount key to achieve high performance is to implement a runtime environment to efficiently schedule the execution of the directed acyclic graph across the multicore platform. This paper studies the impact on the overall performance of some parameters, both at the level of the scheduler (e.g., window size and locality) and the algorithms (e.g., left-looking and right-looking variants). We conclude that some commonly accepted rules for dense linear algebra algorithms may need to be revisited. Copyright © 2011 John Wiley & Sons, Ltd.

Received 3 November 2010; Revised 15 June 2011; Accepted 18 June 2011

KEY WORDS: dense linear algebra; dynamic scheduling; looking variants; multicore architectures

1. INTRODUCTION

The scientific high performance computing community has recently faced dramatic hardware changes with the emergence of multicore architectures. Most of the fastest high performance computers in the world, if not all, mentioned in the last top 500 list [1] released in November 2010 are now based on multicore architectures. This presents the scientific software community with both a daunting challenge and a unique opportunity. The challenge arises from the disturbing mismatch between the design of systems on the basis of this new chip architecture—hundreds of thousands of nodes, a million or more cores, reduced bandwidth, and memory available to cores—and the components of the traditional software stack, such as numerical libraries, on which scientific applications have relied for their accuracy and performance. The state of the art, high performance dense linear algebra software libraries, that is, linear algebra package (LAPACK) [2], have shown limitations on multicore architectures [3]. The performance of LAPACK relies on the use of a standard set of basic linear algebra subprograms (BLAS) [4, 5] within which nearly all of the parallelism occurs following the expensive fork-join paradigm. Moreover, its large stride memory accesses have further exacerbated the problem, and it becomes judicious to efficiently develop existing or new numerical linear algebra algorithms suitable for such hardware.

*Correspondence to: Hatem Ltaief, Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, USA.
†E-mail: Hatem.Ltaief@kaust.edu.sa

As discussed by Buttari *et al.* in [6], a combination of several parameters define the concept of *tile algorithms* and are essential to match the architecture associated with the following cores: (i) fine granularity to reach a high level of parallelism and to fit the core small caches; (ii) asynchronicity to prevent any global barriers; (iii) tile data layout, a high performance data representation to perform efficient memory access; and (iv) dynamic data-driven scheduler to ensure any queued tasks can immediately be processed as soon as all their data dependencies are satisfied.

The parallel linear algebra for scalable multicore architectures (PLASMA) library [7] jointly developed by the University of Tennessee, the University of California Berkeley, and the University of Denver Colorado, tackles this challenge by using tile algorithms to achieve high performance. These tile algorithms can then be represented by directed acyclic graphs (DAGs), where nodes are the tasks and edges are the data dependencies between the tasks. The paramount key is to implement a runtime environment to efficiently schedule the DAG across the multicore platform.

This paper studies the impact on the overall performance of some parameters, both at the level of the scheduler (e.g., window size and locality) and the algorithms, that is, the left-looking (LL) and right-looking (RL) variants of matrix factorization algorithms. The three one-sided factorizations used in this paper represent comprehensive test cases: (i) Cholesky factorization is for symmetric matrices and has a very special DAG of task execution; (ii) LU factorization has to be considered because of the necessary pivoting strategy; and (iii) QR factorization is for nonsymmetric matrices and is characterized by very compute-intensive kernels. The conclusion of this study claims that some commonly accepted rules for dense linear algebra algorithms may need to be revisited.

The remainder of this paper is organized as follows. Section 2 reviews in some detail the mechanisms behind block algorithms (e.g., LAPACK) and explains the different looking variants (LL and RL). Section 3 describes the concepts of tile algorithms. Section 4 introduces the dynamic DAG scheduler and runtime environment. Section 5 shows some performance results. Related work in the area is mentioned in Section 6. Section 7 summarizes the paper and presents future work.

2. BLOCK ALGORITHMS

In this section, we review the paradigm behind the state-of-the-art numerical software, namely, the LAPACK library for shared memory. In particular, we focus on three widely used factorizations in the scientific community, that is, QR, LU, and Cholesky, which are the first steps toward solving numerical linear systems of equations. All the kernels mentioned in the following text have freely available reference implementations as part of either the BLAS or LAPACK libraries.

2.1. Description and Concept

The LAPACK library provides a broad set of linear algebra operations aimed at achieving high performance on systems equipped with memory hierarchies. The algorithms implemented in LAPACK leverage the idea of blocking to limit the amount of bus traffic in favor of a high data reuse. LAPACK consists of a sequential algorithm that relies on parallel BLAS to obtain performance. The level 1 BLAS defines a set of functions related to vector–vector operations, which are memory bound. The level 2 BLAS is characterized by matrix–vector operations. In fact, levels 1 and 2 BLAS correspond to the core routines of LINPACK, the predecessor to LAPACK, where vector machines were dominant in the high performance computing market. With the emergence of systems with memory hierarchies, the concept of blocking for memory capacities was implemented in the level 3 BLAS operations, which consist in matrix–matrix operations. Moreover, block algorithm revolves around an important property of level 3 BLAS operations, the so-called surface-to-volume property, which states that $(\theta(n^3))$ floating point operations are performed on $(\theta(n^2))$ data. Because of this property, level 3 BLAS operations can be implemented in such a way that data movement is limited and reuse of data in the cache is maximized. Block algorithms consist of recasting linear algebra algorithms so that only a small part of the computation is carried out in level 2 BLAS operations (matrix–vector multiplication, where data reuse is limited), whereas most is carried out in level 3 BLAS. Most of these algorithms can be described as the repetition of two fundamental phases as shown in Figure 1.

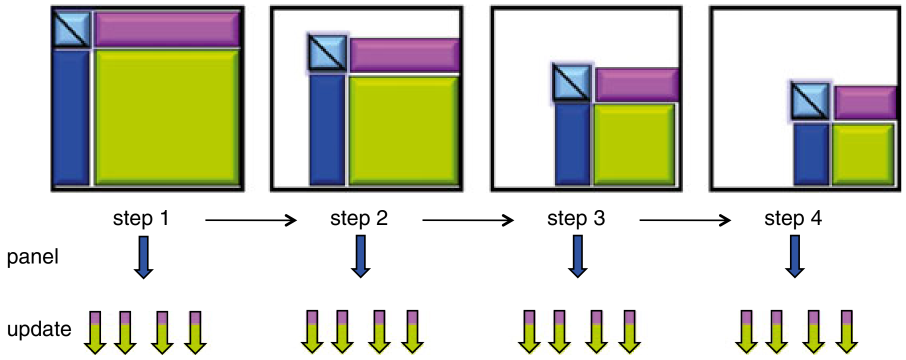


Figure 1. Description and concepts of the LAPACK algorithms.

- *Panel factorization*. Depending on the linear algebra operation that has to be performed, a number of transformations are computed for a small portion of the matrix (*the panel*, the gray and green portion of Figure 1). These transformations, computed using level 2 BLAS operations, can be accumulated.
- *Trailing submatrix update*. In this step, all the transformations that have been accumulated during the panel factorization step have to be applied at once to the rest of the matrix (*the trailing submatrix*, the blue and orange portion of Figure 1) by means of level 3 BLAS operations.

Although the panel factorization can be identified as a sequential execution task that represents a small fraction of the total number of floating point operations per second (required $\theta(n^2)$, for a total of $\theta(n^3)$), the scalability of block factorizations is limited on a multicore system. Indeed, the panel factorization is rich in level 2 BLAS operations that cannot be efficiently parallelized on current shared-memory machines, mainly because of the large stride memory access imposed by the column major data format. Moreover, the parallelism is only exploited at the level of the BLAS routines. This methodology implies a *fork-join* model because the execution flow of a block factorization represents a sequence of sequential operations (panel factorizations) interleaved with parallel ones (updates of the trailing submatrices).

2.2. Block Cholesky factorization

The Cholesky factorization (or Cholesky decomposition) is mainly used as a first step for the numerical solution of linear equations $Ax = b$, where A is symmetric and positive definite. Such systems often arise in physics applications, where A is positive definite because of the nature of the modeled physical phenomenon.

The Cholesky factorization of an $n \times n$ real symmetric positive definite matrix A has the form $A = LL^T$, where L is an $n \times n$ real lower triangular matrix with positive diagonal elements. In LAPACK, the double precision algorithm is implemented by the DPOTRF routine. A single step of the algorithm is implemented by a sequence of calls to the LAPACK and BLAS routines: DSYRK (symmetric rank-k update), DPOTF2 (unblocked Cholesky factorization), DGEMM (general matrix–matrix multiplication), and DTRSM (triangular solver). Because of the symmetry, the matrix can be factorized either as an upper triangular matrix or as a lower triangular matrix.

2.3. Block QR factorization

Generally, a QR factorization of an $m \times n$ real matrix A is the decomposition of A as $A = QR$, where Q is an $m \times m$ real orthogonal matrix and R is an $m \times n$ real upper triangular matrix. QR factorization uses a series of elementary Householder matrices of the general form $H = I - \tau v v^T$, where v is a column reflector and τ is a scaling factor.

Regarding the block algorithms as performed in LAPACK [2] by the DGEQRF routine, nb elementary Householder matrices are accumulated within each panel and the product is represented as

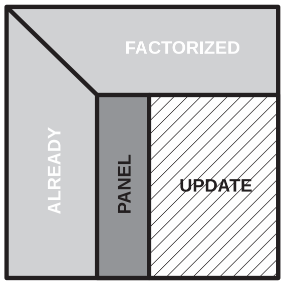
$H_1 H_2 \dots H_{nb} = I - VTV^T$. Here, V is an $n \times nb$ matrix in which columns are the vectors **v**, T is an $nb \times nb$ upper triangular matrix, and nb is the block size. A single step of the algorithm is implemented by a sequence of calls to the LAPACK and BLAS routines: DGEQR2 (panel factorization kernel), DLARFT (computation of the structure T, required for the update kernel), and DLARFB (trailing submatrix update kernel).

2.4. *Block LU factorization*

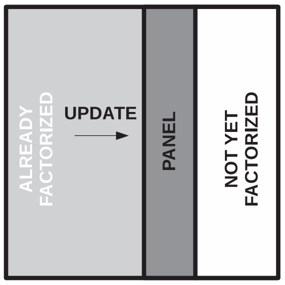
The LU factorization (or LU decomposition) with partial row pivoting of an $m \times n$ real matrix A has the form $A = PLU$, where L is an $m \times n$ real unit lower triangular matrix, U is an $n \times n$ real upper triangular matrix, and P is a permutation matrix. In the block formulation of the algorithm, factorization of nb columns (the panel) is followed by the update of the remaining part of the matrix (the trailing submatrix) [8, 9]. In LAPACK, the double precision algorithm is implemented by the DGETRF routine. A single step of the algorithm is implemented by a sequence of calls to the following LAPACK and BLAS routines: DGETF2, DLASWP (apply pivoting), DTRSM, and DGEMM, where DGETF2 implements the panel factorization and the other routines implement the updates performed in the trailing submatrix.

2.5. *Block looking variants*

Several algorithmic variants exist for the factorizations described previously. The two main ones are called left looking and right looking. They only differ on the location of the update applications with regard to the panel. At each step, the RL variant computes the transformations on the current panel, then it applies those transformations (called *updates*) to the right side on the trailing submatrix. For example, in Figure 2(a), the light gray area represents the portion of the matrix, which has already been factorized. The dark gray area corresponds to the panel, which is currently being factorized. On the right side of the current panel, the dashed area specifies the location of the future updates, once the current panel has been factorized. For the RL variant, the data located in this area is actually transient and may be constantly updated until the end of the whole factorization. In contrast, Figure 2(b) shows the LL variant (also called the 'lazy' variant), which also proceeds by panel, but first updates the current panel by applying all the previous transformations coming from the previous panels (from the left) and then factorizes it. Thus, the updates are not applied to the entire matrix as the RL variant but are limited only to the current panel. The matrix is thus completely factorized one panel at a time. Therefore, the LL variant limits the number of memory accesses while increasing the reuse of the data located on the panel. The LL variant is cache friendly but decreases the parallelism, as the subsequent updates of the remaining matrix columns are delayed and will be eventually applied as the panel computations move forward.



(a) Right Looking Variant.



(b) Left Looking Variant.

Figure 2. Block algorithm looking variants.

2.6. *Limitations of block algorithms*

The block algorithm approach described here is the result of more than a decade of effort and has been considered as a suitable approach for high-end scientific computing. However, the current architecture of multicore chips generally consists of multiple processors located on the same chip. For this architecture, the block algorithm approach has certain limitations because of the presence of coarse-grained tasks and synchronization points between computation steps (because of the *fork-join* paradigm). These limitations seriously affect the efficiency of numerical algorithms, and several research efforts are being undertaken to address this challenge. Because these newer multicore systems require finer granularity and higher asynchronicity, considerable advantage may be obtained by reformulating old algorithms or developing new algorithms in such a way that their implementation can be easily mapped on these new architectures. A number of approaches have been proposed in [6, 10, 11]; block partitioning and hybrid data structures have been studied, and significant performance gains have been demonstrated. Some recent work has focused on looking at operations of the standard LAPACK algorithms, which can be broken into sequences of smaller tasks to achieve finer granularity and higher flexibility when scheduling tasks on processing cores. Algorithms that are developed under this approach are referred to as tile algorithms, and they are described in more detail in the next section.

3. TILE ALGORITHMS

In this section, we describe a solution that removes the fork-join overhead seen in block algorithms. On the basis of tile algorithms, this new model is currently used in shared memory libraries, such as PLASMA and FLAME (University of Texas Austin) [12].

3.1. *Description and concept*

A solution to the fork-join bottleneck in block algorithms has been presented in [6, 10, 11, 13, 14]. The approach consists of breaking the panel factorization and trailing submatrix update steps into smaller tasks that operate on a smaller block. Figure 3 describes how the standard LAPACK algorithm described in Figure 1 is broken into smaller tasks and how synchronization is removed (the panel factorization of step $k + 1$ can start when the first block column of step k has been updated and does not have to wait until the end of the trailing matrix update).

The algorithm can also be represented as a DAG (Figure 4) where nodes represent tasks, either panel factorization or update of a block column, and edges represent data dependencies among them. The execution of the algorithm is performed by asynchronously scheduling the tasks in a way that dependencies are not violated. This asynchronous scheduling results in an out-of-order execution where slow, sequential tasks are hidden among parallel tasks. The following sections describe the tile algorithm paradigm applied to a class of factorizations, that is, Cholesky, LU, and QR, where finer granularity of the operations and higher flexibility for the scheduling can be achieved.

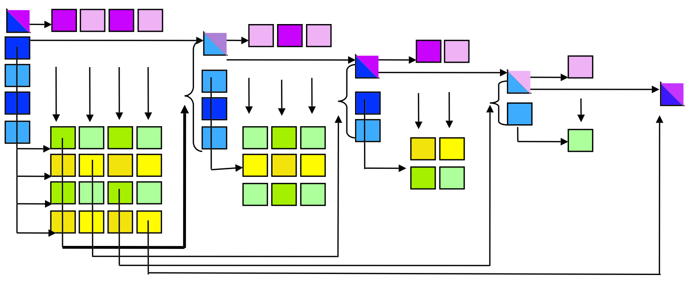


Figure 3. Description and concepts of the tile algorithms.

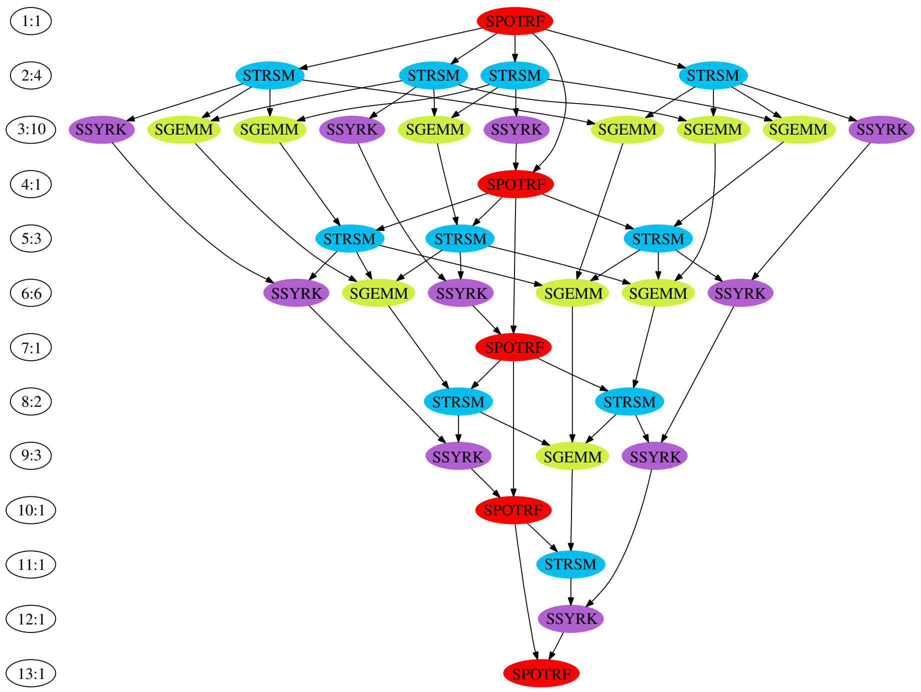


Figure 4. Directed acyclic graphs for a small Cholesky factorization (right-looking version) with five tiles (block size 200 and matrix size 1000). The column on the left shows the depth : width of the directed acyclic graph.

3.2. Tile Cholesky factorization

The tile Cholesky algorithms described in Algorithms 1 (LL variant) and 2 (RL variant) are identical to the block Cholesky algorithm implemented in LAPACK, except for processing the matrix by tiles. Otherwise, exactly the same operations are applied.

Algorithm 1 Left-looking Cholesky

```
1:  for $k = 1, 2$ to NT do
2:      for $i = 1$ to $k$-1 do
3:          {Update $A_{k,k} \leftarrow A_{k,k} - A_{k,i} A_{k,i}^T$}
4:          DSYRK($A_{k,k}$, $A_{k,i}$)
5:      end for
6:      {Cholesky factorization of the tile $A_{k,k}$}
7:      DPOTRF($A_{k,k}$)
8:      for $i = k + 1$ to NT do
9:          for $j = 1$ to $k$ do
10:             {Update $A_{i,k} \leftarrow A_{i,k} - A_{i,j} A_{k,j}$}
11:             DGEMM($A_{i,k}$ $A_{i,j}$, $A_{k,j}$)
12:         end for
13:         {Solve $A_{k,k} X = A_{i,k}$}
14:         DTRSM($A_{k,k}$, $A_{i,k}$)
15:     end for
16: end for
```

3.3. Tile QR factorization

Here, we use a derivative of the block algorithm called the *tile QR* factorization. The tile QR factorization was initially developed to produce a high performance 'out-of-core' implementation [15]

Algorithm 2 Right-looking Cholesky

```
for $k = 1, 2$ to NT do
    {Cholesky factorization of the tile $A_{k,k}$}
    DPOTRF($A_{k,k}$)
    for $i = k + 1$ to NT do
        {Solve $A_{k,k}X = A_{i,k}$}
        DTRSM($A_{k,k}, A_{i,k}$)
        {Update $A_{i,i} \leftarrow A_{i,i} - A_{i,k}A_{i,k}^T$}
        DSYRK($A_{i,i}, A_{i,k}$)
    end for
    for $i = k + 2$ to NT do
        for $j = k + 1$ to $i$ do
            {Update $A_{i,j} \leftarrow A_{i,j} - A_{i,k}A_{j,k}$}
            DGEMM($A_{i,j}$ $A_{i,k}, A_{j,k}$)
        end for
    end for
end for
```

and more recently, to produce a high performance implementation on 'standard' (×86 and alike) multicore processors [6, 13, 16] and on the CELL processor [10].

The algorithm is based on the idea of annihilating matrix elements by square tiles instead of rectangular panels (block columns). The algorithm produces 'essentially' the same R factor as the classic algorithm, for example, as implemented in the LAPACK library (elements may differ in sign). However, a different set of Householder reflectors is produced and a different procedure is required to build the Q matrix. The tile QR algorithm described in Algorithms 3 (LL variant) and 4 (RL variant) relies on four basic operations implemented by four computational kernels:

- CORE_DGEQRT
  The kernel performs the QR factorization of a diagonal tile (situated on the diagonal structure of the matrix) and produces an upper triangular matrix R and a unit lower triangular matrix V containing the Householder reflectors. The kernel also produces the upper triangular matrix T as defined by the compact *WY* technique for accumulating Householder reflectors [17, 18]. The R factor overrides the upper triangular portion of the input, and the reflectors override the lower triangular portion of the input. The T matrix is stored separately.
- CORE_DTSQRT
  The kernel performs the QR factorization of a matrix built by coupling the R factor, produced by CORE_DGEQRT or a previous call to CORE_DTSQRT, with a tile below the diagonal tile. The kernel produces an updated R factor, a square matrix V containing the Householder reflectors, and the matrix T resulting from accumulating the reflectors V. The new R factor overrides the old R factor. The block of reflectors overrides the corresponding tile of the input matrix. The T matrix is stored separately.
- CORE_DORMQR
  The kernel applies the reflectors calculated by CORE_DGEQRT to a tile to the right of the diagonal tile, using the reflectors V along with the matrix T.
- CORE_DSSMQR
  The kernel applies the reflectors calculated by CORE_DTSQRT to two tiles to the right of the tiles factorized by CORE_DTSQRT, using the reflectors V and the matrix T produced by CORE_DTSQRT.

3.4. Tile LU factorization

Here, we use a derivative of the block algorithm called the *tile LU* factorization. Similarly to the tile QR algorithm, the tile LU factorization originated as an 'out-of-core' algorithm [11] and was recently rediscovered for the multicore architectures [13, 16].

Algorithm 3 Left-looking QR

```
for $k = 1, 2$ to NT do
    for $i = 1$ to $k$-1 do
        {Apply transformation on $A_{i,k} \leftarrow (I - V_{ii} T_{ii} V_{ii}^T) A_{i,k}$ }
        CORE_DORMQR($V_{i,i}, T_{i,i}, A_{i,k}$)
        for $j = i + 1$ to NT do
            {Apply transformation on $A_{i,k}$ and $A_{j,k}$}
            CORE_DSSMQR($V_{i,j}, T_{i,j} A_{i,k}, A_{j,k}$)
        end for
    end for
    {QR factorization of the tile $A_{k,k}$}
    CORE_DGEQRT($A_{k,k}, T_{k,k}$)
    for $i = k + 1$ to NT do
        {QR factorization of an upper triangular $A_{k,k}$ and a square $A_{i,k}$}
        CORE_DTSQRT($A_{k,k}, A_{i,k}, T_{i,k}$)
    end for
end for
```

Algorithm 4 Right-looking QR

```
for $k = 1, 2$ to NT do
    {QR factorization of the tile $A_{k,k}$}
    CORE_DGEQRT($A_{k,k}, T_{k,k}$)
    for $i = k + 1$ to NT do
        {Apply transformation on $A_{k,i} \leftarrow (I - V_{kk} T_{kk} V_{kk}^T) A_{k,i}$}
        CORE_DORMQR($V_{i,i}, T_{i,i}, A_{i,k}$)
    end for
    for $i = k + 1$ to NT do
        {QR factorization of an upper triangular $A_{k,k}$ and a square $A_{i,k}$}
        CORE_DTSQRT($A_{k,k}, A_{i,k}, T_{i,k}$)
        for $j = k + 1$ to NT do
            {Apply transformation on $A_{k,i}$ and $A_{k,j}$}
            CORE_DSSMQR($V_{i,k}, T_{i,k} A_{k.j}, A_{i,j}$)
        end for
    end for
end for
```

Again, the main idea here is to annihilate matrix elements by square tiles instead of rectangular panels. The algorithm produces different U and L factors than the standard block algorithm (e.g., the algorithms implemented in the LAPACK library). In particular, we note that the L matrix is not lower unit triangular anymore. Another difference is that the algorithm does not use partial pivoting but a different pivoting strategy. The tile LU algorithm relies on four basic operations implemented by four computational kernels:

- CORE_DGETRF
  The kernel performs the LU factorization of a diagonal tile (situated on the diagonal structure of the matrix) and produces an upper triangular matrix U, a unit lower triangular matrix L and a vector of pivot indexes P. The U and L factors override the input, and the pivot vector is stored separately.
- CORE_DTSTRF
  The kernel performs the LU factorization of a matrix built by coupling the U factor, produced by DGETRF or a previous call to CORE_DTSTRF, with a tile below the diagonal tile. The

kernel produces an updated U factor and a square matrix L containing the coefficients corresponding to the off-diagonal tile. The new U factor overrides the old U factor. The new L factor overrides the corresponding off-diagonal tile. A new pivot vector P is created and stored separately. Because of pivoting, the lower triangular part of the diagonal tile is scrambled and also needs to be stored separately as L'.

- CORE_DGESSM
  The kernel applies the transformations produced by the DGETRF kernel to a tile to the right of the diagonal tile, using the L factor and the pivot vector P.
- CORE_DSSSSM
  The kernel applies the transformations produced by the CORE_DTSTRF kernel to the tiles to the right of the tiles factorized by CORE_DTSTRF, using the L' factor and the pivot vector P.

The tile LU algorithm can then be expressed in a similar way as the tile QR algorithm, described in Algorithms 3 and 4, for the LL and RL variants, respectively, using the numerical LU kernels defined earlier.

One topic that requires further explanation is the issue of pivoting. Because in the tile algorithm only two tiles of the panel are factorized at a time, pivoting only takes place within two tiles at a time, a scheme that could be described as *block-pairwise pivoting*. Clearly, such pivoting is not equivalent to the 'standard' *partial row pivoting* in the block algorithm (e.g., LAPACK). A different pivoting pattern is produced, and because pivoting is limited in scope, the procedure could potentially result in a less numerically stable algorithm. More details on the numerical stability of the tile LU algorithm can be found in [13].

3.5. Tile looking variants

The algorithmic principles of the RL and the LL variants with tile algorithms are similar to block algorithms (Section 2.5). The panel and update regions of the matrix are now split into tiles. The update operations, whether for LL or RL variants, may run concurrently with the panel operations. These variants actually highlight a trade-off between degree of parallelism (RL) and data reuse (LL) and can considerably affect the overall performance. For example, for block algorithms, Cholesky factorization is implemented with the LL variant on shared memory (LAPACK), whereas for distributed memory (ScaLAPACK [19]), the RL variant is used. This paper studies whether this common trade-off rule still holds with tile algorithms on current multicore architectures.

4. DYNAMIC DATA DRIVEN EXECUTION

4.1. Runtime environment for dynamic task scheduling

Restructuring linear algebra algorithms as a sequence of tasks that operate on tiles of data can remove the fork-join bottlenecks seen in block algorithms. This is accomplished by enabling out-of-order execution of tasks, which can hide the work performed by the bottleneck (sequential) tasks. We would like to schedule the sequence of tasks on shared memory, many-core architectures in a flexible, efficient, and scalable manner. In this section, we present an overview of QUARK (QUeuing And Runtime for Kernels) [20], our runtime environment for dynamic task scheduling from the perspective of an algorithm writer who is using the scheduler to create and to execute an algorithm. There are many details about the internals of the scheduler, its dependency analysis, memory management, and other performance enhancements that are not covered here. However, information about an earlier version of this scheduler can be found in [21].

Tasks in this Cholesky factorization example depend on previous tasks if they use the same tiles of data. If these dependencies are used to relate the tasks, then a DAG is implicitly formed by the tasks. A small DAG for a 5×5 tile matrix is shown in Figure 4.

4.2. Determining Task Dependencies

4.2.1. Description of dependency types. For a scheduler to be able to determine dependencies between the tasks, it needs to know how each task is using its arguments. Arguments can be `VALUE`, which are copied to the task, or they can be `INPUT`, `OUTPUT`, or `INOUT`, which have the expected meanings. Given the sequential order that the tasks are added to the scheduler and the way that the arguments are used, we can infer the relationships between the tasks. A task can read a data item that is written by a previous task (read-after-write dependency); a task can write a data item that is written by previous task (write-after-write dependency); or a task can write a data time that is read by a previous task (write-after-read dependency). The dependencies between the tasks form an implicit DAG; however, this DAG is never explicitly realized in the scheduler. The structure is maintained in the way that tasks are queued on data items, waiting to obtain the appropriate access to the data.

4.2.2. From sequential nested-loop code to parallel execution. Our scheduler is designed to use code very similar to the pseudocode described in Algorithm 2. This is intended to make it easier for algorithm designers to experiment with algorithms and design new algorithms. In Figure 5, we can see the final C code from the Cholesky algorithm pseudocode. Each of the calls to the core linear algebra routines is substituted by a call to a wrapper that decorates the arguments with their sizes and their usage (`INPUT`, `OUTPUT`, `INOUT`, `VALUE`). As an example, in Figure 6, we can see how the `DPOTRF` call is decorated for the scheduler.

The tasks are inserted into the scheduler, which stores them to be executed when all the dependencies are satisfied. That is, a task is ready to be executed when all parent tasks have completed. The execution of ready tasks is handled by worker threads that simply wait for tasks to become ready and execute them using a combination of default tasks assignments and work stealing. The thread doing the task insertion, that is, the thread handling the code in Figure 5, is referred to as the master thread. Under certain circumstances, the master thread will also execute computational tasks. Figure 7 provides an idealized overview of the architecture of the dynamic scheduler.

4.2.3. Scheduling a window of tasks. For the applications that we are considering, the number of tasks ($\theta(n^3)$) grows very quickly with the number of `TILES` of data. Figure 8 depicts the growth of the number of tasks when varying the number of tiles. For example, a relatively small LU factorization using 20×20 tiles generates 2870 tasks, whereas using 200×200 tiles generates more than 2.5 millions of tasks. If we were to unfold and retain the entire DAG of tasks for a large problem, we would be able to perform some interesting analysis with respect to DAG scheduling and critical paths. However, the size of the data structures would quickly grow overwhelming. Our solution to this is to maintain a configurable window of tasks. The implicit DAG is then traversed through this sliding window, which should be large enough to ensure that all cores are kept busy. When this window size is reached, the core involved in inserting tasks does not accept any more tasks until some are completed. The usage of a window of tasks has implications in how the loops of an application are unfolded and how much look ahead is available to the scheduler. This paper discusses some of these implication in the context of dense linear algebra applications.

```
for ( i = 0 ; i < p ; i++ ) {
  QUARK_core_dpotrf( quark, 'L', nb[i], A[i][i], nb[i], info );
  for ( j = i+1 ; j < p ; j++ )
    QUARK_core_dtrsm( quark, 'R', 'L', 'T', 'N', nb[j], nb[i],
                  1.0, A[i][i], nb[i], A[j][i], nb[j]);
  for ( j = i+1 ; j < p ; j++ ) {
    for ( k = i+1 ; k < j ; k++ )
      QUARK_core_dgemm( quark, 'N','T', nb[j], nb[k], nb[i],
                   -1.0, A[j][i], nb[j], A[k][i], nb[k],
                    1.0, A[j][k], nb[j]);
    QUARK_core_dpotrf_dsyrk( quark, 'L', 'N', nb[j], nb[i],
                  -1.0, A[j][i], nb[j], +1.0, A[j][j], nb[j]);
  }
}
```

Figure 5. Tile Cholesky factorization that calls the scheduled core linear algebra operations.

```
int QUARK_core_dpotrf( Quark *quark, char uplo, int n,
                       double *A, int lda, int *info )
{
  QUARK_Insert_Task( quark, TASK_core_dpotrf, 0x00,
    sizeof(char),         &uplo,     VALUE,
    sizeof(int),          &n,        VALUE,
    sizeof(double)*n*n,   A,         INOUT | LOCALITY,
    sizeof(int),          &lda,      VALUE,
    sizeof(int),          info,      OUTPUT,
    0);
}
void TASK_core_dpotrf(Quark *quark)
{
  char uplo; int n; double *A; int lda;  int *info;
  quark_unpack_args_5( quark, uplo, n, A, lda, info );
  dpotrf_( &uplo, &n, A, &lda, info );
}
```

Figure 6. Example of inserting and executing a task in the scheduler. The QUARK_core_doptrf routine inserts a task into the scheduler, passing it the sizes and pointers of arguments and their usage (INPUT, OUTPUT, INOUT, VALUE). Later, when the dependencies are satisfied and the task is ready to execute, the TASK_core_dpotrf routine unpacks the arguments from the scheduler and calls the actual dpotrf factorization routine.

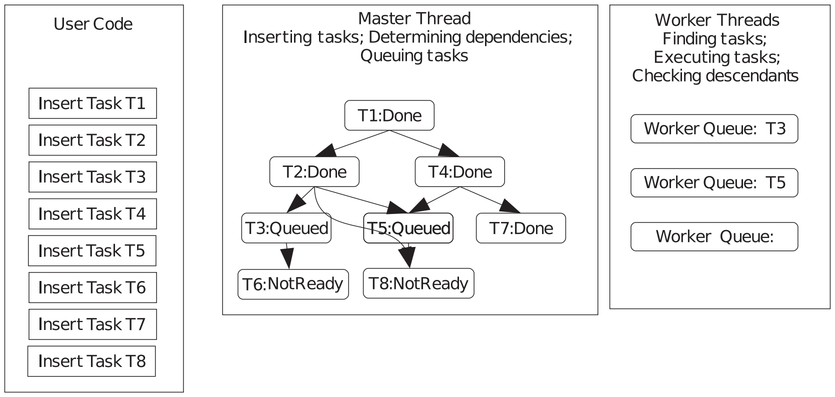


Figure 7. Idealized architecture diagram for the dynamic scheduler. Inserted tasks go into a (implicit) direct acyclic graph based on their dependencies. Tasks can be in NotReady, Queued, or Done states. Workers execute queued tasks and then determine if any descendants have now become ready and can be queued.

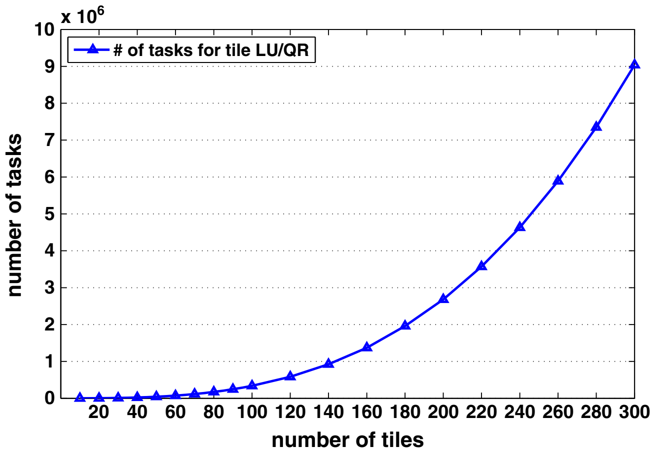


Figure 8. The growth in the number of tasks when increasing the number of tiles. Because the number of tasks grows as $\theta(n^3)$, the entire directed acyclic graph cannot be maintained in memory. The runtime system uses a sliding window of tasks during scheduling and execution.

4.2.4. Data locality and cache reuse. It has been shown in the past that the reuse of memory caches can lead to a substantial performance improvement in execution time. Because we are working with tiles of data that should fit in the local caches on each core, we have provided the algorithm designer with the ability to hint the cache locality behavior. A parameter in a call (e.g., Figure 6) can be decorated with the LOCALITY flag in order to tell the scheduler that the data item (parameter) should be kept in cache if possible. After a computational core (worker) executes that task, the scheduler will assign by default any future task using that data item to the same core. Note that the work stealing can disrupt the by-default assignment of tasks to cores.

The next section studies the performance impact of the locality flag and the window size on the LL and RL variants of the three tile factorizations.

5. EXPERIMENTAL RESULTS

This section describes the analysis of dynamically scheduled tile algorithms for the three factorizations (i.e., Cholesky, QR, and LU) on different multicore systems. The tile sizes for these algorithm have been tuned and are equal to $b = 200$.

5.1. Hardware descriptions

In this study, we consider two different shared memory architectures. The first architecture (system A) is a quad-socket, quad-core machine based on an Intel Xeon EMT64 E7340 processor (Intel, Santa Clara, CA, USA) operating at 2.39 GHz. The theoretical peak is equal to 9.6 Gflop/s per core or 153.2 Gflop/s for the whole node, composed of 16 cores. The practical peak (measured by the performance of a call to a sequential GEMM kernel) is equal to 8.5 Gflop/s per core or 136 Gflop/s for the 16 cores. The level 1 cache, local to the core, is divided into 32 kB of instruction cache and 32 kB of data cache. Each quad-core processor is actually composed of two dual-core Core2 architectures, and the level 2 cache has 2×4 MB per socket (each dual core shares 4 MB). The machine is a non-uniform memory access architecture, and it provides Intel compilers 11.0 and the Intel MKL 10.1 math libraries.

The second system (system B) is an eight-socket, six-core AMD Opteron 8439 SE processor (AMD, Sunnyvale, CA, USA) (48 cores total at 2.8 GHz) with 128 Gb of main memory. Each core has a theoretical peak of 11.2 Gflop/s and the whole machine 537.6 Gflop/s. The practical peak (measured by the performance of a call to a sequential GEMM kernel) is equal to 9.5 Gflop/s per core or 456 Gflop/s for the 48 cores. There are three levels of cache. The level 1 cache consists of 64 kB, and the level 2 cache consists of 512 kB. Each socket is composed of six cores, and the level 3 cache has 6 MB 48-way associative shared cache per socket. The machine is a non-uniform memory access architecture, and it provides Intel Compilers 11.1 and the Intel MKL 10.2 math libraries.

5.2. Performance discussions

In this section, we evaluate the effect of the window size and the locality feature on the LL and RL tile algorithm variants.

The nested loops describing the tile LL variant codes are naturally ordered in a way that already promotes locality on the data tiles located on the panel. Figure 9 shows the effect of the locality flag of the scheduler on the overall performance of the tile LL Cholesky variant. As expected, the locality flag does not really improve the performances when using small window sizes. The scheduler is indeed not able to perform enough lookahead to anticipate the reuse occurring in the next panels. With larger window sizes, the scheduler is now able to acquire knowledge of data reuse in the next panels and takes full advantage of it. Therefore, the locality scheduler flag permits the performance of the LL tile Cholesky variant to increase by up to 5%–10% for large window sizes. The LL tile QR and LU behave in the same manner as the LL tile Cholesky. On the contrary, the RL variants for the three factorizations are not affected by this parameter. The RL variant triggers on the right side of the panel so many parallel independent tasks that the chance for any data reuse opportunities is significantly decreased.

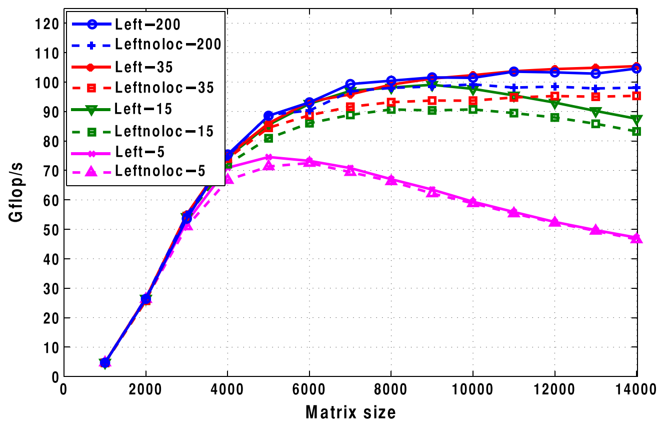


Figure 9. Cholesky factorization (left-looking variant) on system A, 16 threads: effect of the scheduler data locality parameter.

The next parameter to be optimized for the three tile factorizations is the window size of the scheduler. Figure 10 shows how critical the optimization of this parameter can be. The figure displays the execution trace of the tile LL Cholesky algorithm with small and large scheduler window sizes on the system A. We observe that for a small window size (bottom trace), the execution is stretched because there are not enough queued tasks to feed the cores, which makes them turn to an ‘idle’ state (white spaces). This is further demonstrated by looking at the performance graphs from Figures 11–13 obtained on both systems A and B for the three factorizations. The performance is represented as a function of the matrix size. In these figures, the size of the window is defined by the number indicated in the legend times the number of the available threads. By increasing the size of the scheduler window, the performances of the LL variants of the three factorizations considerably increase. For example, the overall performance of the tile LL Cholesky

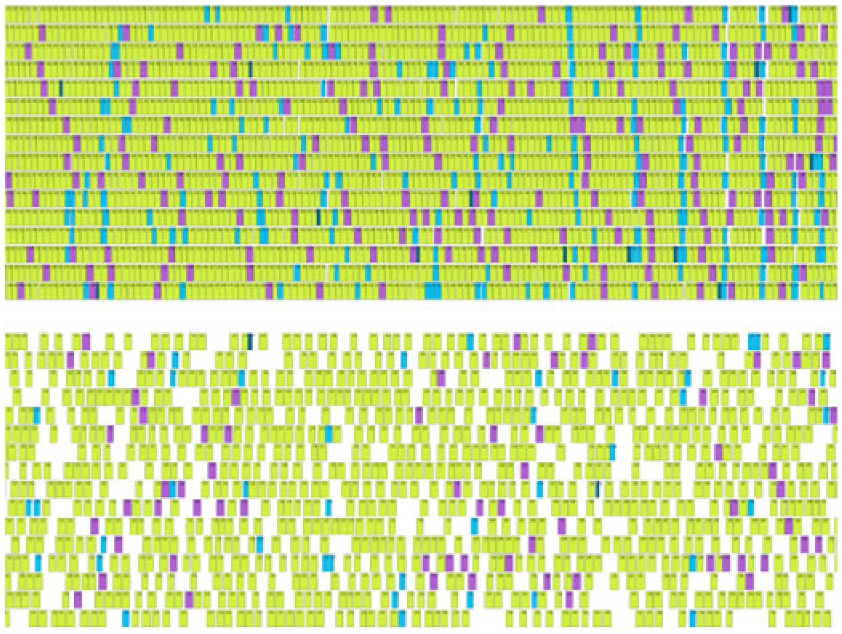
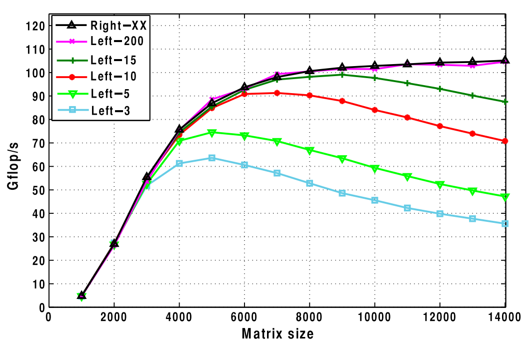
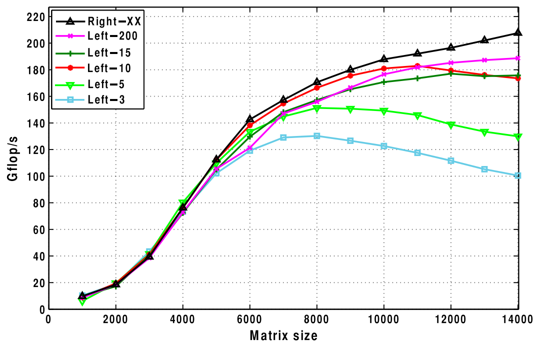


Figure 10. A snapshot of the execution trace of Cholesky factorization (left-looking variant) consisting of LAPACK and BLAS calls to DSYRK (purple), DPOTRF (dark blue), DGEMM (green), and DTRSM (blue). Using a large window of tasks (top) versus small window of tasks (bottom) shows how the task window size can affect lookahead and thus, the scheduling trace and performance.

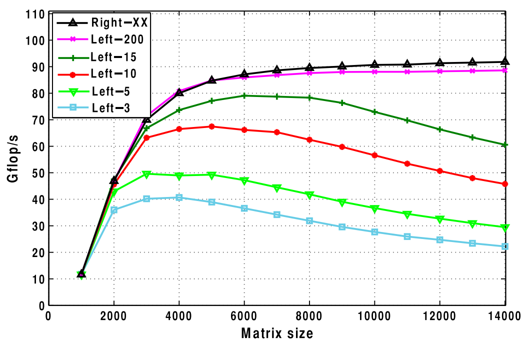


(a) System A: 16 threads

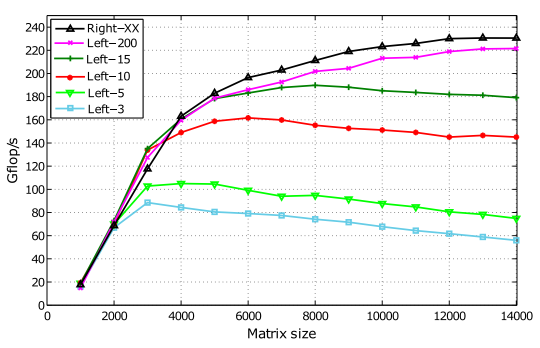


(b) System B: 48 threads

Figure 11. Cholesky factorization (left-looking versus right-looking variants) with different task window size.

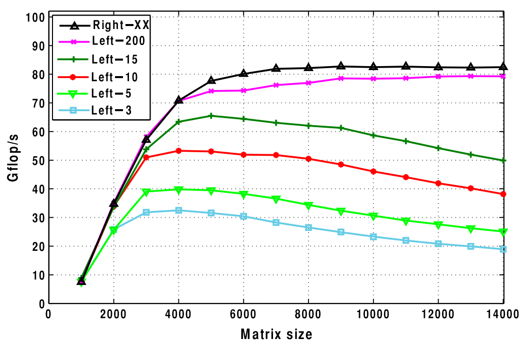


(a) System A: 16 threads

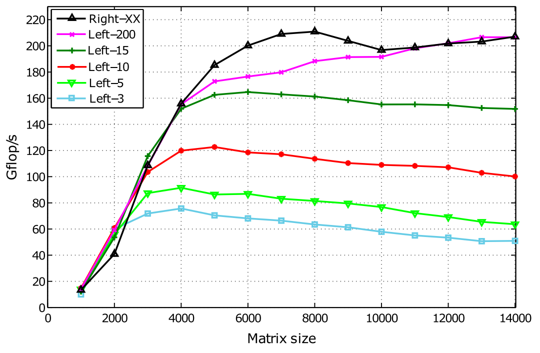


(b) System B: 48 threads

Figure 12. QR factorization (left-looking versus right-looking variants) with different task window size.



(a) System A: 16 threads



(b) System B: 48 threads

Figure 13. LU factorization (left-looking versus right-looking variants) with different task window size on 48 threads.

is multiplied by 3 on system A and by 2 on system B when increasing the window size per core from 3 to 200. In the same way, the performance of the tile LL QR is multiplied by 4 on systems A and B. The performance of the tile LL LU is also multiplied by 3 when increasing the window size of the scheduler on both systems A and B. So, in other words, if the scheduler window size is small, the runtime environment system is not able to detect and anticipate the concurrent execution of independent sets of tasks. The already low performance curve encountered with small window sizes on the three factorizations starts significantly dropping even more for large matrix sizes.

On the other hand, the window size parameter has no impact on the tile RL variants. The performance curves of the RL Cholesky, QR, and LU denoted by $Right - XX$ outperform the LL ones regardless of the window sizes. The curves eventually meet up for large enough window sizes. Again, this is due to the high degree of parallelism offered by the RL variants, which allows many tasks to be simultaneously executed. The RL variants seem to be very attractive for shared-memory multicore systems, especially because they do not require any parameter auto-tuning like the LL variants. Therefore, the RL variants should be chosen by default in the PLASMA library distribution for those tile algorithms.

Furthermore, by closely studying some execution traces of LL and RL variants, the clear distinctions between both variants inherited from block algorithms (i.e., LAPACK) may not exist anymore in the context of data-driven, asynchronous out-of-order DAG execution. Indeed, by increasing the window sizes of the scheduler, the tile LL variants are able to perform lookahead techniques by initiating the work on the next panel, whereas the one on the current panel is still pending. Likewise, the tile RL variants are permitted to start processing the next panel while the updates of the current panel are still ongoing. We speculate that in the future, using a dynamic runtime execution environment will cause the distinctions between these algorithmic variant to be blurred.

6. RELATED WORK

The FLAME project [12] developed by the University of Texas, Austin, follows the same algorithmic principle by splitting the matrix into finer blocks. However, the runtime environment (SuperMatrix) requires the explicit construction of the DAG before the actual parallel execution of the tasks. As depicted in Figure 8, the size of a DAG considerably increases with respect to the number of tiles. This may necessitate a large amount of memory to allocate the data structure which is not an option for scheduling large matrix sizes, especially when dealing with other algorithms (e.g., two-sided transformations) that generate a more complex and larger DAG compared with QR, LU, and Cholesky.

Furthermore, there are many projects that are designed to provide high performance, near-transparent computing environments for shared-memory machines. Here, we discuss two projects that have been considered during the design of our runtime environment.

The SMP superscalar project[22][23] from the Barcelona Supercomputing Center is a programming environment for shared-memory, multicore architectures focused on the ease of programming, portability, and flexibility. A standard C or Fortran code can be marked up using preprocessor pragma directives to enable task level parallelism. The parameters to the functions are marked as input, output, or inout, and the data dependencies between tasks are inferred to determine a task DAG. A source-to-source compiler and a supporting runtime library are used to generate native code for the platform. The SMP superscalar project shares many similarities with the dynamic runtime environment presented here. One difference is that our implementation uses an application programming interface to express parallelism rather than compiler pragmas, thus eliminating an intermediate step. Another difference is that our runtime environment allows for specialized flags, such as the LOCALITY flag, which enables tuning for linear algebra algorithms.

The Cilk project [24] [25] from the MIT Laboratory for Computer Science is a compiler-based extension of the C language that gives the programmer a set of keywords to express task level parallelism (*cilk, spawn, sync, inlet, abort*). Cilk is well suited to algorithms that can be expressed recursively and implements a fork-join model of multithreaded computation. Because the parallelism in our algorithms is expressed in a DAG obtained through a data dependency analysis, Cilk is not well suited to our problems.

7. CONCLUSION AND FUTURE WORK

This paper presents an analysis of dynamically scheduled tile algorithms for dense linear algebra on shared-memory, multicore systems. In particular, the paper highlights the significant impact of the locality feature and the scheduler task-window size on the LL and RL variants of the tile Cholesky, QR, and LU. The tile RL variants of the three factorizations outperform the LL ones regardless of the scheduler locality and window size optimizations. It is a common rule of thumb that LL variants of linear algebra algorithms (e.g., Cholesky) are well suited to shared-memory architectures, and RL variants are better suited to distributed memory architectures. However, this is no longer valid in the context of data-driven, asynchronous out-of-order DAG execution environments. Algorithms that expose more parallelism (the RL variants) can now perform better in a shared-memory multicore environment than algorithms designed mainly for locality and cache awareness (the LL variants). This asynchronous DAG execution can morph one algorithmic variant into another, for example, the use of large task-window sizes enables the LL variants to have sufficient lookahead opportunities to expose similar levels of parallelism as the RL variants and thus achieve similar levels of performance. The difference between the LL and RL variants of the algorithms has been blurred in this environment.

ACKNOWLEDGEMENTS

The research reported here was partially supported by the National Science Foundation and the Microsoft Research.

REFERENCES

1. TOP500 Supercomputing Sites. (Available from: http://www.top500.org/).
2. Anderson E, Bai Z, Bischof C, Blackford LS, Demmel JW, Dongarra JJ, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D. *LAPACK Users' Guide*. SIAM: Philadelphia, PA, 1992. (Available from: http://www.netlib.org/lapack/lug/).
3. Agullo E, Hadri B, Ltaief H, Dongarrra J. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *Sc '09: Proceedings of the conference on high performance computing networking, storage and analysis*. ACM: New York, NY, USA, 2009; 1–12.
4. Dongarra JJ, Croz JD, Duff IS, Hammarling S. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)* 1990; **16**:1–17.
5. Lawson CL, Hanson RJ, Kincaid D, Krogh FT. Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software (TOMS)* 1979; **5**:308–323.
6. Buttari A, Langou J, Kurzak J, Dongarra JJ. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice & Experience* 2008; **20**(13):1573–1590.
7. PLASMA users' guide, parallel linear algebra software for multicore architectures, version 2.0, University of Tennessee, 2009.
8. Demmel JW. *Applied Numerical Linear Algebra*. SIAM, 1997.
9. Dongarra JJ, Duff IS, Sorensen DC, van der Vorst HA. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.
10. Kurzak J, Dongarra JJ. QR factorization for the CELL processor. *Scientific Programming* 2009; **17**(1–2):31–42.
11. Quintana-Ortí ES, van de Geijn RA. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software (TOMS)* 2008; **35**(2):11.
12. Van Zee FG, Chan E, van de Geijn RA, Quintana-Orti ES, Quintana-Orti G. The `libflame` Library for Dense Matrix Computations. *Computing in Science and Engineering* 2009; **11**(6):56–63. DOI: http://doi.ieeecomputersociety.org/10.1109/MCSE.2009.207.
13. Buttari A, Langou J, Kurzak J, Dongarra JJ. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing Systems Application* 2009; **35**:38–53.
14. Kurzak J, Buttari A, Dongarra J. Solving systems of linear equations on the CELL processor using Cholesky factorization. *IEEE Transactions on Parallel Distributed Systems* 2008.
15. Gunter BC, van de Geijn RA. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software* 2005; **31**(1):60–78.
16. Chan E, Quintana-Orti ES, Quintana-Orti G, van de Geijn R. Supermatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. *Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'07*, 2007; 116–125.
17. Bischof C, van Loan C. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing* 1987; **8**:2–13.
18. Schreiber R, van Loan C. A storage-efficient WY representation for products of Householder transformations. *Journal on Scientific and Statistical Computing* 1991; **10**:53–57.
19. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra JJ, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC. *ScaLAPACK Users' Guide*. SIAM: Philadelphia, PA, 1997. (Available from: http://www.netlib.org/scalapack/slug/).
20. YarKhan A, Kurzak J, Dongarra J. QUARK users' guide: QUeueing And Runtime for Kernels, Technical Report, ICL-UT-11-02, Innovative Computing Laboratory, University of Tennessee, 2011.

21. Kurzak J, Dongarra J. Fully dynamic scheduler for numerical scheduling on multicore processors. *Technical Report LAWN (LAPACK Working Note) 220, UT-CS-09-643*, Innovative Computing Lab, University of Tennessee, 2009.
22. SMP Superscalar (SMPSs) User's Manual, Version 2.0, Barcelona Supercomputing Center, 2008.
23. Perez JM, Badia RM, Labarta J. A dependency-aware task-based programming environment for multi-core architectures. *Cluster'08*, 2008; 142–151.
24. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP)*: Santa Barbara, California, 1995; 207–216.
25. Cilk 5.4.6 reference manual, Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science, 2001.